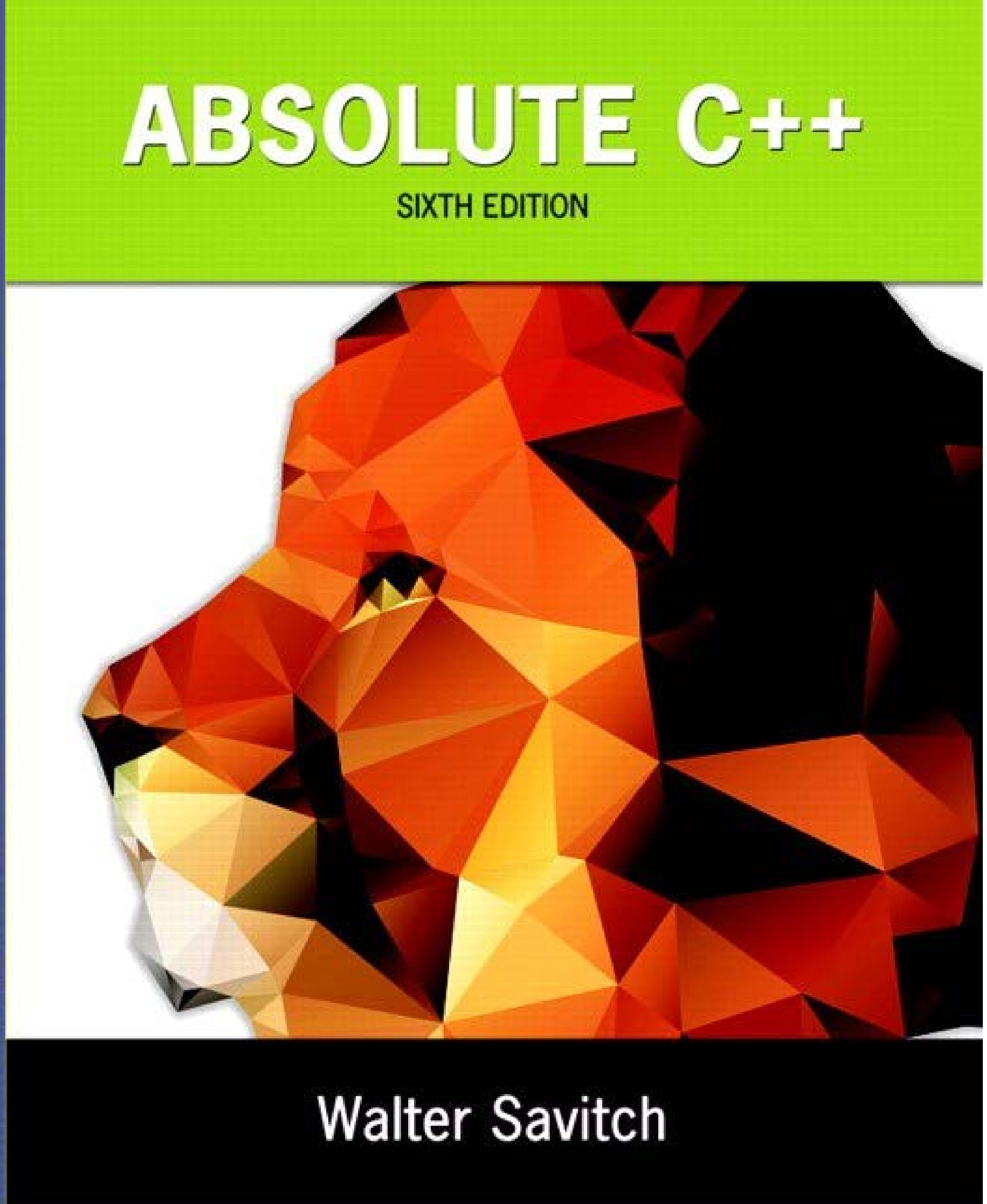


# Chapter 11

## Separate Compilation and Namespaces

PEARSON

## Learning Objectives

- Separate Compilation
  - Encapsulation reviewed
  - Header and implementation files
- Namespaces
  - using directives
  - Qualifying names
  - Unnamed namespaces
  - Hiding helping functions
  - Nested namespaces

# Separate Compilation

- Program Parts
  - Kept in separate files
  - Compiled separately
  - Linked together before program runs
- Class definitions
  - Separate from "using" programs
  - Build library of classes
    - Re-used by many different programs
    - Just like predefined libraries

## Class Separation

- Class Independence
  - Separate class definition/specification
    - Called "interface"
  - Separate class implementation
  - Place in two files
- If implementation changes → only that file need be changed
    - Class specification need not change
    - "User" programs need not change

## Encapsulation Reviewed

- Encapsulation principle:
  - Separate how class is used by programmer from details of class's implementation
- "Complete" separation
  - Change to implementation → NO impact on any other programs
- Basic OOP principle

## Encapsulation Rules

- Rules to ensure separation:
  1. All member variables should be private
  2. Basic class operations should be:
     - Public member functions
     - Friend or ordinary functions
     - Overloaded operators

     Group class definition and prototypes together
     - Called "interface" for class
  3. Make class implementation unavailable to users of class

## More Class Separation

- Interface File
  - Contains class definition with function and operator declarations/prototypes
  - Users "see" this
  - Separate compilation unit
- Implementation File
  - Contains member function definitions
  - Separate compilation unit

## Class Header Files

- Class interface always in header file
  - Use .h naming convention
- Programs that use class will "include" it
  - #include "myclass.h"
  - Quotes indicate you wrote header
    - Find it in "your" working directory
  - Recall library includes, e.g., <iostream>
    - < > indicate predefined library header file
    - Find it in library directory

## Class Implementation Files

- Class implementation in .cpp file
  - Typically give interface file and implementation file same name
    - myclass.h and myclass.cpp
  - All class's member function defined here
  - Implementation file must #include class's header file
- .cpp files in general, typically contain executable code
  - e.g., Function definitions, including main()

## Class Files

- Class header file #included by:
  - Implementation file
  - Program file
    - Often called "application file" or "driver file"
- Organization of files is system dependent
  - Typical IDE has "project" or "workspace"
    - Implementation files "combined" here
    - Header files still "#included"

# Multiple Compiles of Header Files

- Header files
  - Typically included multiple times
    - e.g., class interface included by class implementation and program file
  - Must only be compiled once!
  - No guarantee "which #include" in which file, compiler might see first
- Use preprocessor
  - Tell compiler to include header only once

## Using #ifndef

- Header file structure:

  -
    ```
    #ifndef FNAME_H
    #define FNAME_H
    … //Contents of header file
    …
    #endif
    ```
- FNAME typically name of file for consistency, readability
- This syntax avoids multiple definitions of header file

## Other Library Files

- Libraries not just for classes
- Related functions
  - Prototypes → header file
  - Definitions → implementation file
- Other type definitions
  - structs, simple typedefs → header file
  - Constant declarations → header file

## Namespaces

- Namespace defined:
  A collection of name definitions
  - Class definitions
  - Variable declarations
- Programs use many classes, functions
  - Commonly have same names
  - Namespaces deal with this
  - Can be "on" or "off"
    - If names might conflict → turn off

## using Directive

- using namespace std;
  - Makes all definitions in std namespace available
- Why might you NOT want this?
  - Can make cout, cin have non-standard meaning
    - Perhaps a need to redefine cout, cin
  - Can redefine any others

## Namespace std

- We've used namespace std
- Contains all names defined in many standard library files
- Example:
  #include <iostream>
  - Places all name definitions (cin, cout, etc.) into std namespace
  - Program doesn't know names
  - Must specify this namespace for program to access names

# Global Namespace

- All code goes in some namespace
- Unless specified → global namespace
  – No need for using directive
  – Global namespace always available
  – Implied "automatic" using directive

## Multiple Names

- Multiple namespaces
  - e.g., global, and std typically used
- What if name defined in both?
  - Error
  - Can still use both namespaces
  - Must specify which namespace used at what time

# Specifying Namespaces

- Given namespaces NS1, NS2
  - Both have void function myFunction() defined differently

```
{
    using namespace NS1;
    myFunction();
}
{
    using namespace NS2;
    myFunction();
}
```

  - using directive has block-scope

# Creating a Namespace

- Use namespace grouping:

```
namespace Name_Space_Name
{
      Some_Code
}
```

- Places all names defined in Some_Code into namespace Name_Space_Name
- Can then be made available:

```
using namespace Name_Space_Name
```

## Creating a Namespace Example

- Function declaration:

```
namespace Space1
{
        void greeting();
}
```

- Function definition:

```
namespace Space1
{
        void greeting()
        {
                cout << "Hello from namespace Space1.\n";
        }
}
```

## using Declarations

- Can specify individual names from namespace
- Consider:
  Namespaces NS1, NS2 exist
  Each have functions fun1(), fun(2)
  - Declaration syntax:
    using Name_Space::One_Name;
  - Specify which name from each:
    using NS1::fun1;
    using NS2::fun2;

# using Definitions and Declarations

- Differences:
  - using declaration
    - Makes ONE name in namespace available
    - Introduces names so no other uses of name are allowed
  - using directive
    - Makes ALL names in namespace available
    - Only "potentially" introduces names

## Qualifying Names

- Can specify where name comes from
  - Use "qualifier" and scope-resolution operator
  - Used if only intend one use (or few)
- NS1::fun1();
  - Specifies that fun() comes from namespace NS1
- Especially useful for parameters:
  int getInput(std::istream inputStream);
  - Parameter found in istream's std namespace
  - Eliminates need for using directive or declaration

## Naming Namespaces

- Include unique string
  - Like last name
- Reduces chance of other namespaces with same name
- Often multiple programmers write namespaces for same program
  - Must have distinct names
  - Without → multiple definitions of same name in same scope
    - Results in error

# Class Namespace Example: **Display 11.6** Placing a Class in a Namespace (Header File)

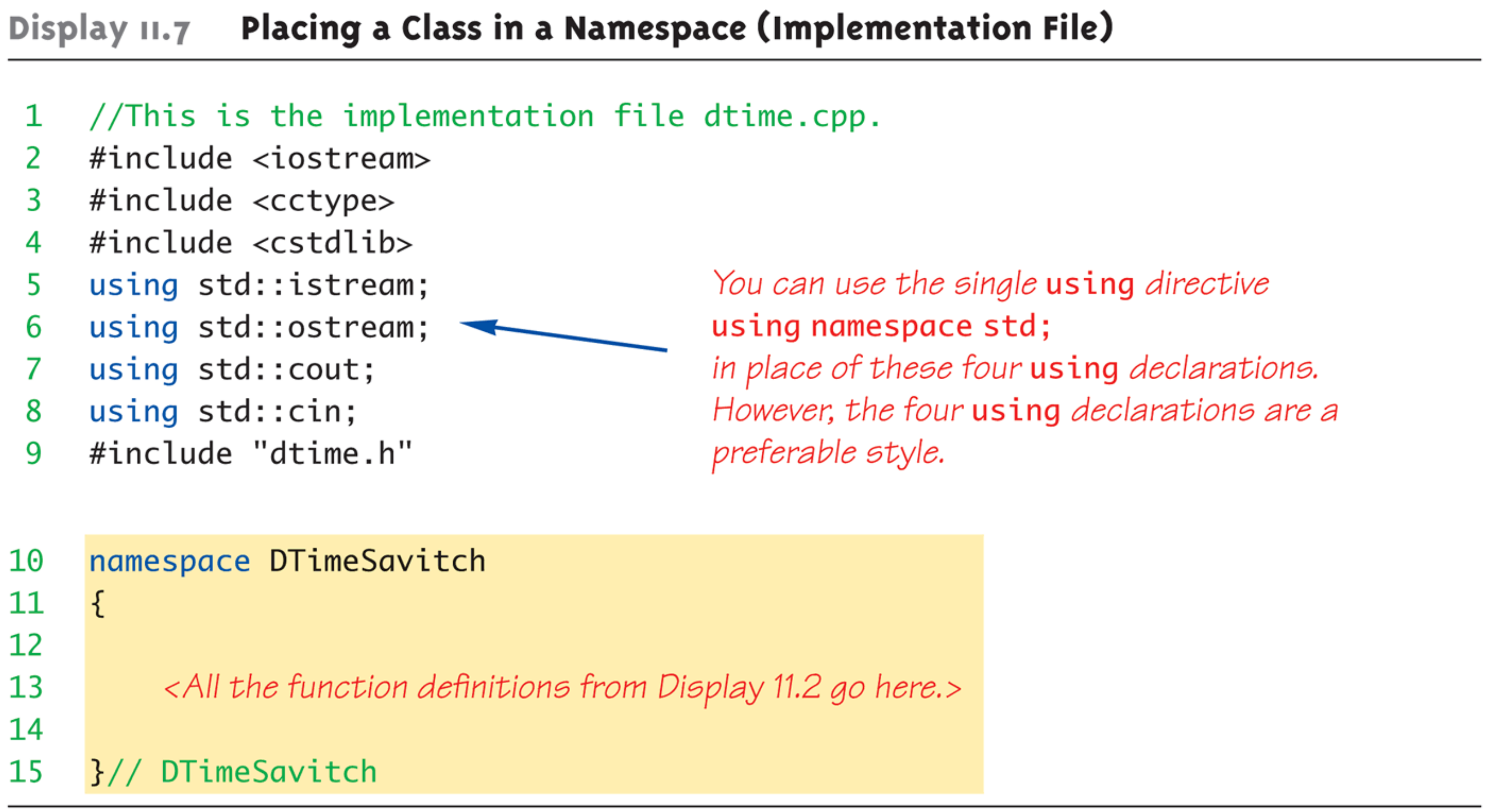
Display 11.6 **Placing a Class in a Namespace (Header File)**

```
//This is the header file dtime.h.
#ifndef DTIME_H
#define DTIME_H

#include <iostream>
using std::istream;
using std::ostream;

namespace DTimeSavitch
{

    class DigitalTime
    {

       <The definition of the class DigitalTime is the same as in Display 11.1.>
    };

}// DTimeSavitch

#endif //DTIME_H
```

*A better version of this class definition will be given in Displays 11.8 and 11.9.*

*Note that the namespace* **DTimeSavitch** *spans two files. The other is shown in Display 11.7.*

## Class Namespace Example: **Display 11.7** Placing a Class in a Namespace (Implementation File)



**Display 11.7 Placing a Class in a Namespace (Implementation File)**

```
//This is the implementation file dtime.cpp.
#include <iostream>
#include <cctype>
#include <cstdlib>
using std::istream;
using std::ostream;
using std::cout;
using std::cin;
#include "dtime.h"

namespace DTimeSavitch
{

    <All the function definitions from Display 11.2 go here.>

}// DTimeSavitch
```

*You can use the single* `using` *directive* `using namespace std;` *in place of these four* `using` *declarations. However, the four* `using` *declarations are a preferable style.*

# Unnamed Namespaces

- Compilation unit defined:
  - A file, along with all files #included in file
- Every compilation unit has unnamed namespace
  - Written same way, but with no name
  - All names are then local to compilation unit
- Use unnamed namespace to keep things "local"
- Scope of unnamed namespace is compilation unit

## Global vs. Unnamed Namespaces

- Not same
- Global namespace:
  - No namespace grouping at all
  - Global scope
- Unnamed namespace:
  - Has namespace grouping, just no name
  - Local scope

# Nested Namespaces

- Legal to nest namespaces

```
namespace S1
{
      namespace S2
      {
            void sample()
            {
                  …
            }
      }
```

- Qualify names twice:
  - S1::S2::sample();

# Hiding Helping Functions

- Recall helping function:
  - Low-level utility
  - Not for public use
- Two ways to hide:
  - Make private member function
    - If function naturally takes calling object
  - Place in class implementation's unnamed namespace!
    - If function needs no calling object
    - Makes cleaner code (no qualifiers)

## Summary 1

- Can separate class definition and implementation → separate files
  - Separate compilation units
- Namespace is a collection of name definitions
- Three ways to use name from namespace:
  - Using directive
  - Using declaration
  - Qualifying

## Summary 2

- Namespace definitions are placed inside namespace groupings
- Unnamed namespace
  - Used for local name definitions
  - Scope is compilation unit
- Global namespace
  - Items not in a namespace grouping at all
  - Global scope